2019

Towards an assessment framework of reuse: A Knowledge Level Analysis Approach

Ghassan Beydoun
University of Technology Sydney, beydoun@uow.edu.au

Achim Hoffmann
University of New South Wales, achim@cse.unsw.edu.au

Rafael Valencia-Garcia
University of Murcia

Jun Shen
University of Wollongong, jshen@uow.edu.au

Asifqumer Gill
University of Technology Sydney

Towards an assessment framework of reuse: A Knowledge Level Analysis Approach

Abstract

The process of assessing the suitability of reuse of a software component is complex. Indeed, software systems are typically developed as an assembly of existing components. The complexity of the assessment process is due to lack of clarity on how to compare the cost of adaptation of an existing component versus the cost of developing it from scratch. Indeed, often pursuit of reuse can lead to excessive rework and adaptation, or developing suites of components that often get neglected. This paper is an important step towards modelling the complex reuse assessment process. To assess the success factors that can underpin reuse, we analyze the cognitive factors that belie developers' behavior during their decision-making when attempting to reuse. This analysis is the first building block of a broader aim to synthesize a framework to institute activities during the software development lifecycle to support reuse.

Keywords

level, knowledge, analysis, reuse:, approach, framework, assessment, towards

Disciplines

Engineering | Science and Technology Studies

Publication Details

Beydoun, G., Hoffmann, A., Valencia-Garcia, R., Shen, J. & Gill, A. (2019). Towards an assessment framework of reuse: A Knowledge Level Analysis Approach. Complex and Intelligent Systems, Online first 1-9.

Complex & Intelligent Systems
https://doi.org/10.1007/s40747-019-0116-1

ORIGINAL ARTICLE

Towards an assessment framework of reuse: a knowledge-level analysis approach

Ghassan Beydoun[1] · Achim Hoffmann[2] · Rafael Valencia Garcia[3] · Jun Shen[4] · Asif Gill[1]

Received: 11 May 2018 / Accepted: 24 June 2019

Abstract
The process of assessing the suitability of reuse of a software component is complex. Indeed, software systems are typically developed as an assembly of existing components. The complexity of the assessment process is due to lack of clarity on how to compare the cost of adaptation of an existing component versus the cost of developing it from scratch. Indeed, often pursuit of reuse can lead to excessive rework and adaptation, or developing suites of components that often get neglected. This paper is an important step towards modelling the complex reuse assessment process. To assess the success factors that can underpin reuse, we analyze the cognitive factors that belie developers' behavior during their decision-making when attempting to reuse. This analysis is the first building block of a broader aim to synthesize a framework to institute activities during the software development lifecycle to support reuse.

Keywords Reuse · Reuse process modelling · Software development ontology · Components' reuse · Ontologies

Introduction

It is well accepted that successful software components' reuse can reduce the development cost and time of the software. Context aware adaptation [13] and software reuse can lead to shorter coding time and more reliable code. These expected benefits were strong motives in the reuse research that has been ongoing since the late 1960s. It continued with the take up of OO technology in 80s and 90s [22], software patterns [10, 11], the interest in Service Oriented Architecture (SOA) [29], and current interest in Open Source platforms such as GitHub [27]. Reuse often seems an appealing economical path to software development, but it is fraught with disappointing pitfalls that hinder economic reuse from taking place. For example, a web developer may attempt to adapt a component which seems initially suitable, only to find after substantial time adapting it, that starting from scratch was easier and quicker. Whilst much work has been accomplished, there is still much to do before the software development via reuse is completely achieved [5, 12].

Reuse may cause unforeseen maintenance problems; for example, a database developer may be tempted to reuse a data retrieval component which later turns out to be very memory hungry and constantly causes his database to crash. Little effort has gone into understanding the actual mental processes involved in reuse and the factors that render it a success or a failure. Towards this, in this paper, we propose an abstract model of the process of software component reuse and seek to identify the reuse factors that underlie the decisions in this process. We apply a knowledge-level analysis to the process of reuse. This analysis yields an abstract description of the reuse process. This description also identifies features of reusable components and the factors at play in the process.

✉ Ghassan Beydoun
ghassan.beydoun@uts.edu.au

Achim Hoffmann
achim@unsw.edu.au

Rafael Valencia Garcia
valencia@um.es

Jun Shen
jshen@uow.edu.au

Asif Gill
asif.gill@uts.edu.au

1 University of Technology Sydney, Faculty of Engineering and IT, City Campus, Sydney, Australia

2 School of Computer Science and Engineering, University of New South Wales,, NSW 2052 Kensington, Australia

3 Faculty of Computer Science, University of Murcia, Murcia, Spain

4 School of Computing and Information Technology, University of Wollongong, 2522 Wollongong, NSW, Australia

Published online: 11 July 2019

Towards operationalization of the result of this analysis, we sketch an ontology-based approach to index and represent the knowledge required. In particular, to enable the use of these features and development of an appropriate repository, details are elaborated on using an ontology that can be used to index reusable components.

The rest of this paper is organised as follows: Sect. 2 presents the overarching perspective that applies to the reuse assessment framework. Section 3 presents an example illustrating the processes identified in Sect. 2 to enable further elaboration of the reuse factors. Section 4 presents the reuse factors highlighting how they can be described at the knowledge level. Section 5 presents how the overall approach can be operationalised with appropriate ontologies to interface to reusable components. Section 6 concludes and summarizes the contributions of the paper.

Modelling assessment of reuse process

Majority of software development projects rely heavily on reuse. The systems themselves are often developed from an assembly of existing reusable components [4]. Whether reuse is one of the reusing components by developers or project management strategies by a senior manager, the reuse process as we will see involves iterative decomposition and adaptation of knowledge artefacts [15]. To assess the reuse of components effectively, it is necessary to measure the reusability of these components [30], although the length and the complexity of reuse process steps may depend on the domains and the reusable artefact. For example, a project manager may spend longer time assessing the reuse potential of a development methodology than what a software developer would spend in evaluating a software component.

In formulating our framework, we first separate the knowledge involved from the processes involved in reuse. We model the actual process of reuse as an iterative process cycle interleaving analysis/retrieval/reuse/decomposition (see Fig. 1). Some examples of knowledge involved in this process are component retrieval knowledge, component adaptation knowledge, component reuse knowledge, and component construction knowledge. The availability of these and other kinds of knowledge impact the transition between the various activities involving reuse. We identify the various types of knowledge involved in reuse and present guidelines to assess the viability of reuse using knowledge complexity measures applied to the various kinds of knowledge involved. For example, reuse involves locating a library component and then adapting it. If either the knowledge and/or effort involved in locating and/or adapting the component is too much to afford, then reuse may not be viable. As we develop this framework at the knowledge level, we model the actual reuse process which

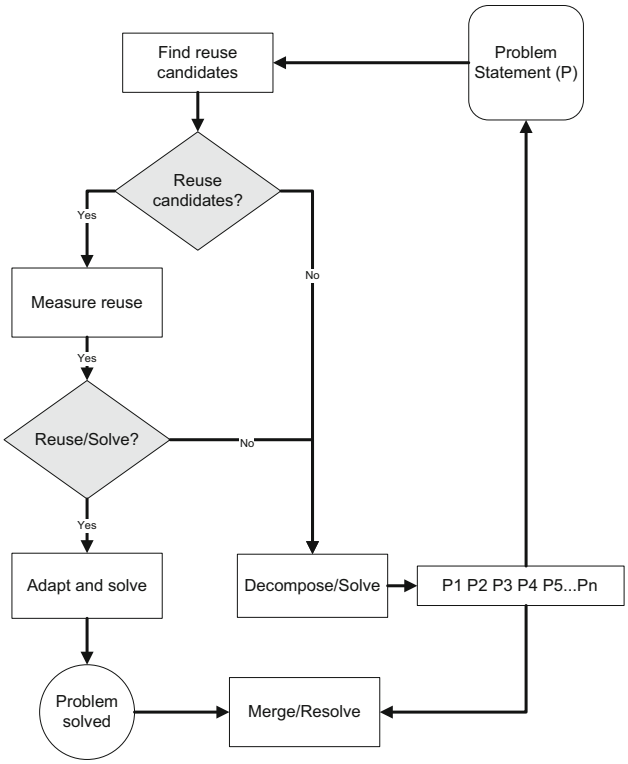

Fig. 1 The iterative reuse process—initial problem statement is examined. Ideally, a reusable solution is adapted and problem solved. More likely, the problem is decomposed into smaller problems and a partial solution via reuse is possible. Reuse may fail in two circumstances: if adaptation fails or if merging with the rest of the emerging solution fails

seems to us to be a domain independent process. This is in contrast with the domain dependence of the complexity of the actual components to be reused, as results in [12, 22] suggest.

Our separation between reuse process and knowledge being reused is also useful in identifying success factors in reuse. The process factors can also be construed as developer-related factors, whilst the knowledge being reused in practice reflects the nature of the components reused and how they are organised (i.e., the reuse libraries). We delve into both sets of factors and delineate negative factors that hinder a successful and economical reuse from taking place, and positive factors facilitating a successful and economical reuse. We also identify various relations between these factors.

Factors that emanate from the nature of knowledge reused include the following:

- Factors related to the software components (required) e.g., how complex the component is and has this component been already reused?
- Factors that are related to the domain of the component; e.g., how application area specific is the component?
-

Factors related to the libraries used, e.g., how well documented they are. E.g., if the access to the components is obscured due to bad documentation, then reuse may suffer.

Factors that emanate from the reuse process:

- Factors that are related to the software development lifecycle (SDLC), e.g., are reuse encouraged or frowned upon. For example, a scrum master of a safety critical application development team may dictate the level of reuse depending on certain security requirements.
- Factors that are related to the developers involved, e.g., do they wear programming (rather than reuse) as a badge of honour? In an open source environment, some developers may wish to leave their mark by avoiding reuse.

All those factors will be further analyzed and operationalised in our model. Not all lend themselves to a knowledge-level analysis with the same ease. E.g., component complexity can be seen from a knowledge complexity perspective only if we see a piece of software as an embodiment of some knowledge and in itself as a knowledge artefact.

A case study: a web-based application to sell health insurance policies

In this section, we present a reuse episode in a software development example to highlight the features of our framework and the reuse process modelled highlighted in the previous section. We first describe the requirements of the software pursued.

A developer is tasked with providing a web-based interface to sell health insurance policies. The selling process involves the web customer entering the various options associated with a policy as well as entering their personal details and payment options. The developer is reasonably familiar with an e-commerce shopping trolley web component which instantly comes to their mind as they think about the requirements of the interface. The shopping trolley component is typically used by websites like Amazon.com where more than one product is bought in one e-shopping session. In what follows, we illustrate the pitfalls that reusing the shopping trolley may present when developing a web interface to sell insurance policy with various option. We also analyze the knowledge involved/required/missing in the various stages of this reuse episode.

The developer decomposes the functionalities of the shopping trolley into two: collect products and collect payments (register functionality). He maps these two to collecting insurance options and collect insurance payments and surmises that since the shopping trolley is an available component which seems comprehensible, it seems worth reusing. In other words, the process of assessing to reuse or not is hierarchical and involves decomposing the functionalities of the artefact to be reused. In support of the process described in Fig. 1, there are sub-processes assessing the artefact against the decomposition of the problem (Fig. 2). We overview the adaptation of each of these two functionalities to the requirements of the web-based insurance policies which leads the developer reassessing the suitability of the shopping trolley.

A health insurance policy is unlike a book or a flower bouquet; it is pointless for a person to have more than one of them. Government regulations may also well imply that an insurance company can only sell one to any one person. This creates additional requirements for the interface: (1) authenticating the identity of the customer to ensure that they do not already have an existing policy, (2) if they have one and they are pursuing additional options only, then, in this case, the interface requires identity authentication too, and (3) a connection to a database of existing customers is required.

An insurance policy is often a hierarchical product. A kind of insurance is chosen; for example, a hospital cover or a dental cover. When the broad category is chosen, various options are made available. In other words, the ideal representation is a hierarchy of windows presenting the various options at various stages during the sale. This is not possible with a shopping trolley. A trolley simply collects the products in one set, so strictly speaking the initial view of the developer that an option equates to a product is a compromise which will restrict the functionalities provided by their interface.

From this simple episode of reuse attempt, these kinds of knowledge were evident and have impact on the final outcome of the development:

- Knowledge of the shopping trolley may have been short in not seeing immediately that it does not provide hierarchical representation or authentication of buyers.
- Knowledge of alternative choices may have motivated the initial desire to use the shopping trolley.
- Initial knowledge of the domain may have thwarted the desire to reuse the shopping trolley.
- Technical knowledge interfacing/development may ultimately decide which way the developer goes with the shopping trolley.

And finally, perhaps the knowledge embedded in the shopping trolley itself and how complicated it is to reuse will have an impact on the final outcome.

Characterising factors surrounding reuse

We know that certain components, such as databases, are regularly and frequently reused. We know also that not all attempts to reuse turn out to be successful. Therefore, the

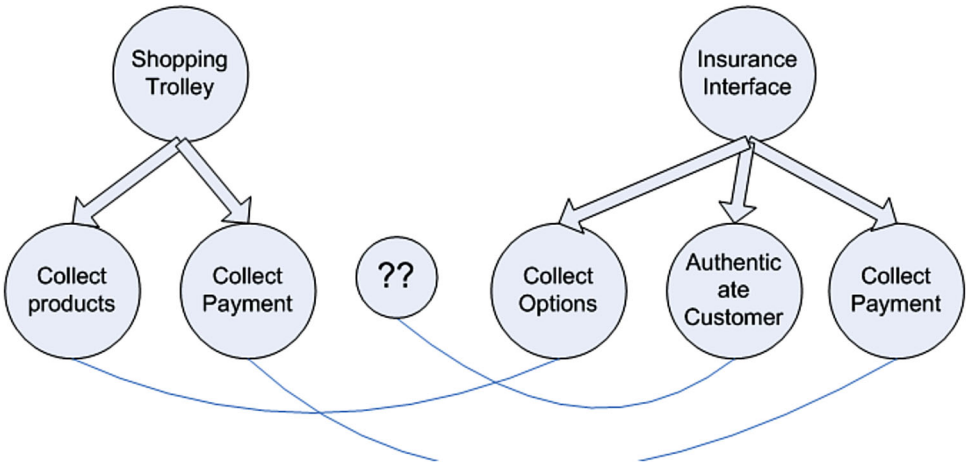

Fig. 2 A decomposition of the artefact to be reused parallels the problem decomposition to assess whether to adapt and reuse or to solve the problem independently. Knowledge of artefact and that of problem domain are directly involved. Knowledge of alternatives and technical knowledge of developer are indirectly involved

right balance needs to be found. Too many components in the library make the task of selecting a suitable component difficult. Too few components render the library also relatively useless. However, it appears that not only that the general fact that a component could be reused makes it fit to be placed in the component library, but also the degree to which it is easy to determine whether a given component can be effectively reused for a given project. In other words, a key feature of a component to be reused is whether it can be readily identified as suitable for a project or otherwise dismissed without any undue effort. This entails on one hand that the capabilities and limitations of a library component should be easy and quick to grasp for a system designer/developer. On the other hand, even if the functionality is easy to grasp, there should only be a rather minimalistic set of components that need to be considered for a given project. For example, numerous slight variations of a given base function are probably not worth having in a library as the variations might be considered many times and end up being dismissed anyway. Therefore, they would consume considerable time of the system designers/developers without contributing accordingly. In the odd situation where a given function requires some slight modification, it might be much easier to then use the base version of the function and develop some wrapper function around it from scratch to meet the requirements.

Developer-related factors

The developer's context and requirements are clearly a primary determinant. This leads to their search of a reusable component in the first place. There is evidence that some application domain knowledge on the developer's side may be helpful [6, 26] to the ultimate outcome to reuse or not to reuse. Indeed, this is not too surprising as this may lower the cognitive costs involved in reuse. The application domain knowledge determines the nature of their initial plan idea, initial code ideas, and so forth. In other words, identifying the most suitable component to be reused seems to be a knowledge-driven task. This knowledge comes from experience and components require proper description to be identified properly [25, 28]. On the other hand, in some cases, programmers' eagerness to prove themselves can be an obstacle to reuse. What seems clear though is that there are issues specific to code reuse (e.g., code easy to reproduce and replicate) and many general reuse issues applicable to code reuse as well (e.g., requiring experience).

SDLC-related factors

Reuse can also be seen from the perspective of the development project managers. The project manager's knowledge of the team, the repository availability, and any later maintenance are all determinants to the uptake or success of reuse. Some of these reuse issues are common across many domains (not only in software engineering) and in relations to many artefacts, e.g., car designs, building components. For instance, some process actions are impacted across team processes. Reuse may require additional communication within the team to ensure that the interfaces between components are feasible or economical to develop. Reuse may impact software developers who may be tempted (or avoiding) to reuse components are (or not) familiar with. But specific to software perhaps, reuse can also influence early phases of the development, e.g., the requirements gathering (for example, a client may be tempted to request extensions to their system).

Component-related factors

The success of reuse is also based on the features of the components available themselves. Components that are easy to locate or to adjust to fit the current context are more likely to be reused. The component reuse cost depends on the complexity of entity reused. The more complex the entity is to

re-write from scratch, the more likely it is that it will be successfully reused; the less complex it is the less justifiable the above costs are. Furthermore, the more frequently needed a component is, the more developing from scratch is justified as the overhead cost per component is reduced.

Library-related factors

Successfully locating a component which may be reusable also depends on the repository itself. This can be quite hard in some instances, e.g., trying to locate a reusable function in a large file of code. Hence, availability and interfaces of the repository of components are key factors. The maintenance of the repository is also a longer term factor. This depends on the cohesion of the repository, the representation of components, and as well as the knowledge about components. More storage can render people untidy and lead to increase in the complexity of libraries and their reuse. However, broadening access can also enhance reuse abut which will require additional protocols for keeping libraries tidy and user friendly. More web-based interfaces such as GitHub aim towards this [9].

One has to wonder what kind of component library would really be useful. To characterise when reuse is economical, we first consider the theoretical upper bound on the cost of reuse. Suppose that there is a library containing all possible functions of some limited complexity. The size of this library requires that a complex search is undertaken to find a function, where the search complexity outweighs the complexity of the function to be retrieved. In this case, reuse can never be justified. However, in reality, reuse does occur, and libraries of code are used. This indicates that the complexity of reused components does indeed exceed the complexity of finding the component within the library. Considering a theoretical case where a library for reuse is available that contains all possible source code programs up to a certain length. Then selecting the correct function from that library would pose a serious problem. One simple access key would actually be to specify the code itself. This would obviously not help at all. The other end of the spectrum though would contain no function in the library at all, which would also be useless.

Operationalization of reuse framework with ontologies

As discussed above, one of the most important issues in software reuse is storage and retrieval from the libraries of components that will be reused. In fact, searching for information is one of the most dominant daily activities of the participating developers and the retrieval of reusable assets requires an appropriate description [20, 27]. The libraries used typically have some descriptions of their functionality in natural language and do not incorporate formalized knowledge about application domains or the functionality of the artefacts [2, 3, 19]. Several traditional keyword-based search methods have been proposed in last years, but they have not been very successful and found their shortfalls in finding software assets for reuse [18]. The use of semantic web technologies provides better representation mechanisms for components that can be described by means of metadata. Some different approaches of providing metadata to software artefacts have been proposed. For example, in the work presented in [24], a General Process Ontology to annotate common and general software process are proposed. Business process modelling has also been represented by means of ontologies and ontology-based tools to support the graphical modelling of business processes with information derived from domain ontologies [8]. In [19], ontologies are used to describe software artefacts independently from a particular format.

Assessing quickly to what degree a given reusable component will match the requirements of a project is critically important for enabling effective reuse on a larger scale. While proper documentation of the capabilities and limitations of components would be expected, current practice is certainly falling behind those expectations. More importantly, though, it is generally difficult to describe capabilities and limitations if those future potential uses have not been considered so far. E.g., in the example above, the designers of the shopping trolley modules would not have considered the specific needs for selling health insurance policies online. We propose to represent the components by means of ontologies. An ontology can be defined as "a formal and explicit specification of a shared conceptualization" [31, 34]. Ontologies provide a formal, structured knowledge representation, with the advantage of being reusable and shareable. Ontologies provide a common vocabulary for a domain and define with different levels of formality—the meaning of the terms and the relations between them. Knowledge in ontologies is mainly formalized using five kinds of components: classes, relations, attributes, axioms, and instances. Classes in the ontology are usually organised into taxonomies. Sometimes, the definition of ontologies has been diluted, in the sense that taxonomies are considered to be full ontologies [31].

The ontology that describes the components has to take into account different types of knowledge (Fig. 3) such as knowledge about the complexity, types and reuse of software components, knowledge about the functionality of the component, and knowledge related to the domain of the component. The software artefact description ontology models the different types of software artefacts (specification, diagrams, databases schemas, software components, libraries, web services, etc.) and the complexity of each component and some reuse measures.

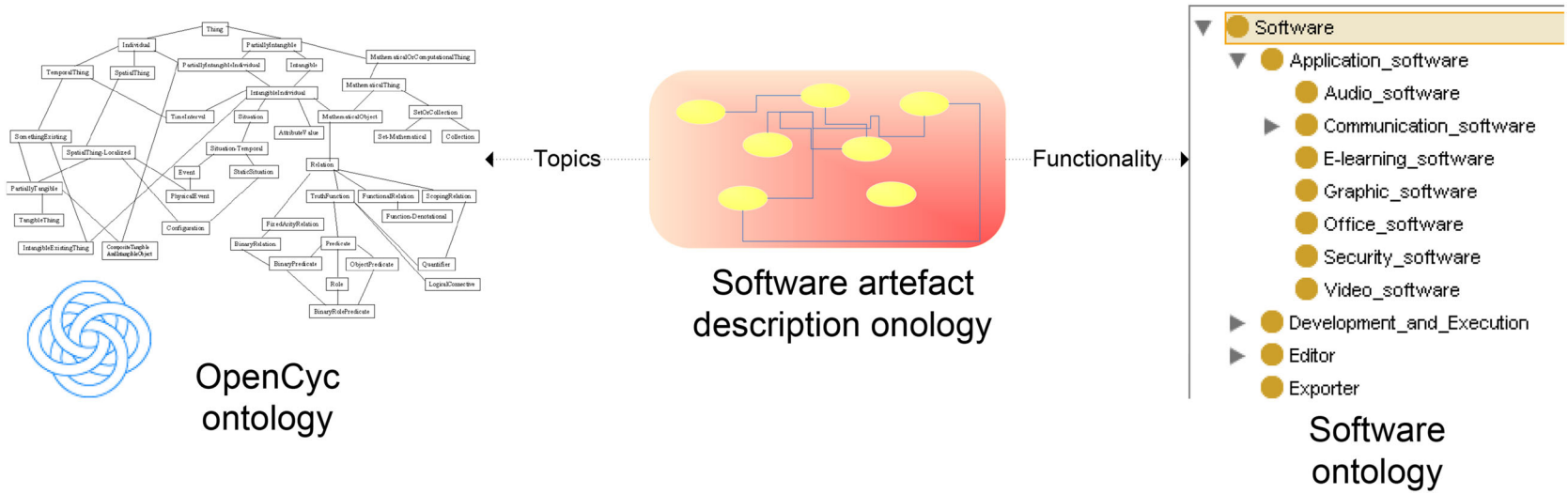

Fig. 3 The knowledge represented in the description of a software artefact

Fig. 4 The knowledge included in the description of a software artefact

That is, rather than hoping that documentation of developed software will eventually improve and allow more effective reuse, we are advocating the need for interactive tools that assist in quickly assessing the suitability of existing software components for a given project.

The functionality of the software component and the domain where this artefact has been used (i.e., finances, e-commerce, etc.) would be represented with annotations to other ontologies such as a software ontology or a general ontology, respectively.

The development of an ontology for a particular domain is a time-consuming task, while a shared vocabulary is necessary for obtaining an agreed ontology. The software ontology would represent a vocabulary of software engineering representing the functionality of the software component. Some ontologies and vocabularies for Software engineering have been developed in the last few years. These ontologies have been applied in different steps of software engineering lifecycle and these approaches can be classified in four areas [19]:

- Ontology-driven development subsumes the usage of ontologies at development time that describes the problem domain itself.
- Ontology-enabled development also uses ontologies at development time, but for supporting developers with their tasks.
- Ontology-based architectures use an ontology as a primary artefact at runtime. The ontology makes up a central part of the application logic.

- Ontology-enabled architectures (OEA), finally, leverage ontologies to provide infrastructure support at the run time of a software system.

For the purposes of this work, the second category of ontologies is required. That is, ontologies that semantically describe the functional properties of software artefacts can be reused. A representative example within this area is shown in [23], in which an ontology for requirements specification documents has been developed and used for modelling reusable security requirements. In [21], the ontology-based DESWAP system is presented. In the context of the DESWAP project, a knowledge base with comprehensive semantic descriptions of software and their functionalities were developed. Thus, by taking into account the shortcomings of developing a new ontology from scratch, the ontologies developed under the scope of the DESWAP project for representing the features and functional properties of the software projects have been adapted and reused. An excerpt of the software ontology representing a classification of software is shown in Fig. 4.

Discussion and conclusion

Re-users often cannot articulate a proper description of the component that they are seeking. This research also aims at looking how to take an incomplete articulation of what is sought and develop it through a sequence of interactions to zoom in on a reusable component, or in a worst case scenario ascertain that the possibility of effective reuse is not available. This work will help developers to select the best component in terms of its reusability, which will improve the maintainability of the overall system [30]. The adoption and reuse of the components can also be guided and justified by the higher level emerging technology themes [16], reference models [14], and design patterns [17]. For instance, technology themes can provide the overall high-level trends and context as starting point for identifying the needs for required components. This can be further assisted by the selected reference model (e.g., cloud reference model) to classify the components and their relationship such as the relevant software application (e.g., SaaS) and platform (e.g., PaaS) components within the layered architecture paradigm. Furthermore, design patterns (e.g., MVC) can describe the configuration of the software components for a context with the view to enable reusability of components and their integration with other components for similar contexts or problems. Thus, the concepts of technology themes, reference models, and design patterns may provide the additional description for enhancing the reusability or adaptation of the components for a specific context and problem.

The proposed framework is an explanatory tool. It can help in explaining failure and success stories or reuse. It identifies critical thresholds of success and failure, e.g., domain knowledge in some cases is essential for effective reuse, a minimal component complexity is required to justify reuse. The model is also a predictive tool that can point to the potential pitfalls and caveats in pursuing or broadening the scope of reuse, e.g., how to best represent libraries, how to best avail interfaces in reuse of the form web services or applying a P2P community-based searching for components [32, 33], and how to best interact with the user to ensure that a component that can be reused effectively once it is found.

Knowledge Management in Software Engineering has attracted significant interest in the research community as well as in industrial practice [1, 7]. However, the work which we are aware of has essentially focussed on specific knowledge relevant to the organisation and concerning the general software design process as opposed to issues discussed in this paper. The work illustrated how domain knowledge can be stored for reuse and how a reuse process can be enacted. The enactment of the process within a development project was sketched in the reuse of a shopping trolley component, but it has been yet illustrated in an actual development project. This will be the focus of a collaboration with an industry partner (Surround Australia Pty Ltd).

References

1. Aurum A, Daneshgar F, Ward J (2008) Investigating knowledge management practices in software development organisations—an Australian experience. Inf Softw Technol 50(2008):511–533
2. Beydoun G, Hoffmann A (1998) Simultaneous modelling and knowledge acquisition using NRDR. PRICAI'98 Top Artif Intell, pp 83–95
3. Beydoun G, Hoffmann A (2001) Theoretical basis for hierarchical incremental knowledge acquisition. Int J Hum Comput Stud 54(3):407–452
4. Beydoun G, Tran N, Low G, Henderson-Sellers B (2005) Foundations of ontology-based MAS methodologies. Agent Oriented Inf Syst 3:111–123
5. Beydoun G, Hoffmann A, Breis JTF, Bejar RM, Valencia-Garcia R (2005) Cooperative modelling evaluated. Int J Coop Inf Syst 14(01):45–71
6. Beydoun G, Low G, Tran N, Bogg P (2011) Development of a peer-to-peer information sharing system using ontologies. Expert Syst Appl 38(8):9352–9364
7. Beydoun G, Low G, García-Sánchez F, Valencia-García R (2014) Identification of ontologies to support information systems development. Inf Syst 46:45–60

8. Born M, Dörr F, Weber I (2007) User-friendly semantic annotation in business process modeling. Lecture notes in Computer Science 4832 Springer 2007, pp 260–271
9. Brown R, Beydoun G, Low G, Tibben W, Zamani R, García-Sánchez F (2016) Computationally efficient ontology selection in software requirement planning. Inf Syst Front 18(2):349–358
10. Buschmann F, Eunier RM, Rohnert H, Sommerland P, Stahl M, Stahl M (1996) Pattern-oriented software architecture volume 1: a system of patterns. Wiley, Chichester
11. Do TT, Kolp M, Faulkner S, Pirotte A (2004) Introspecting agent-oriented design patterns. In: Chang SK (ed) Advances in software engineering and knowledge engineering. World Publishing, Singapore, pp 151–177
12. Frakes WB, Kang K (2005) Software reuse research: status and future. IEEE Trans Softw Eng 31(7):529–536
13. Gill AQ, Bunker D (2011) Conceptualization of a context aware cloud adaptation (CACA) framework. In: 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing (pp 760–767). IEEE
14. Gill AQ, Smith S, Beydoun G, Sugumaran V (2014) Agile enterprise architecture: a case of a cloud technology-enabled government enterprise transformation. In: Proceeding of the 19th Pacific Asia Conference on Information Systems (PACIS 2014) (pp 1–11). United States: AISeL
15. Gill AQ (2015) Adaptive cloud enterprise architecture. World Scientific, Singapore
16. Gill AQ, Bunker D, Seltsikas P (2015) Moving forward: emerging themes in financial services technologies' adoption. Commun Assoc Inf Syst 36:12
17. Gill AQ, Phennel N, Lane D, Phung VL (2016) IoT-enabled emergency information supply chain architecture for elderly people: the Australian context. Inf Syst 58:75–86
18. Hadji H, Choi H (2009) Towards contextual information based-approach to support software reuse system," In: Proceedings of the 11th international conference on Advanced Communication Technology. Gangwon-Do, South Korea, pp 132–136
19. Happel H, Korthaus A, Seedorf S, Tomczyk P, KOntoR (2006) An ontology-enabled approach to software reuse. In: Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering (SEKE), San Francisco, July 2006, pp 349-354
20. Happel H, Schuster T, Szulman P (2008) Leveraging source code search for reuse. Lecture notes in computer science 5030 Springer 2008, pp 221–232
21. Hartig O, Kost M, Freytag J-C (2008) Designing component-based semantic web applications with DESWAP. In: Proceedings of the Poster and Demonstration Session at the 7th International Semantic Web Conference (ISWC), Karlsruhe, Germany, Oct. 2008
22. Henderson-Sellers B (1996) Object-oriented metrics. Measures of complexity. Prentice Hall, Washington
23. Lasheras J, Valencia-García R, Fernández-Breis JT, Toval A (2009) 2009 Modelling reusable security requirements based on an ontology framework. J Res Pract Inf Technol 41(2):119–133
24. Lin Y, Strasunskas D (2005) Ontology-based semantic annotation of process templates for reuse. In: Proceedings of 10th International workshop EMMSAD 2005, Porto, Portugal, pp 162–167
25. Lopez-Lorca A, Beydoun G, Valencia-Garcia R, Martinez-Bejar R (2016) Supporting agent oriented requirement analysis with ontologies. Int J Hum Comput Stud 87:20–37
26. Miller T, Lu B, Sterling L, Beydoun G, Taveter K (2014) Requirements elicitation and specification using the agent paradigm: the case study of an aircraft turnaround simulator. IEEE Trans Softw Eng 40(10):1007–1024
27. Oliveira J, Fernandes E, Vale G, Figueiredo E (2017) Identification and prioritization of reuse opportunities with Jreuse. In: International Conference on Software Reuse (ICSR 2017): Mastering Scale and Complexity in Software Reuse pp 184–191
28. Othman SH, Beydoun G (2011) A disaster management metamodel (DMM) validated. In: Kang BH., Richards D. (eds) Knowledge Management and Acquisition for Smart Systems and Services. Springer, Berlin, Heidelberg, pp 11–125
29. Shashwar A, Kumar D (2017) A service identification model for service oriented architecture. In: 2017 3rd International Conference on Computational Intelligence and Communication Technology (CICT), IEEE
30. Sharma A, Grover PS, Kumar R (2009) Reusability assessment for software components. ACM SIGSOFT Softw Eng Notes 34(2):1–6
31. Studer R, Benjamins R, Fensel D (1998) Knowledge engineering: principles and methods. Data knowl Eng 25:161–197
32. Tran N, Low G, Beydoun G (2006) A methodological framework for ontology centric agent oriented software engineering. Comput Syst Sci Eng 21(2):117
33. Tran N, Beydoun G, Low G (2007) Design of a peer-to-peer information sharing MAS using MOBMAS (ontology-centric agent oriented methodology). In: Wojtkowski W, Wojtkowski WG, Zupancic J, Magyar G, Knapp G (eds) Advances in Information Systems Development. Springer, Boston, MA, pp 63–76
34. Xu D, Wijesooriya C et al (2011) Outbound logistics exception monitoring: a multi-perspective ontologies' approach with intelligent agents. Expert Syst Appl 38(11):13604–13611